Parsing All of C
by Taming the Preprocessor

Robert Grimm, New York University
Joint Work with Paul Gazzillo
Genesis
(1969-1973)
And God Said
Let There Be C
Genesis
(1976, 1986, and 1987)
And God Said
Let There Be

emacs, gdb, and gcc
And It Was Very Good
And It Was Very Good
The Good

• Operating systems
  • iOS, OS X, Linux, Free/Open/Net BSD

• Databases
  • SQLite, Berkeley DB, IBM DB2

• Internet servers
  • Apache, Bind, Sendmail
The Bad

> emacs main.c
> gcc main.c
> ./a.out
> gdb a.out
We Need Better Tools

• Source browsing

• Bug finding

• Automated refactoring
We Need Better Tools

- Source browsing
  - Cxref, LXR
- Bug finding
  - Astrée, Coverity
- Automated refactoring
  - Apple’s Xcode, Eclipse CDT
Genesis (1969-1973)
The Serpent Said
You Will Be Like God With

The Preprocessor
The Fall From Grace

• Preprocessor adds concision, expressivity, and abstraction
  • Conditional compilation

• Macros

• File inclusion
The Fall From Grace

- Preprocessor adds concision, expressivity, and abstraction
  - Conditional compilation
    - Code has many meanings
  - Macros
- File inclusion
The Fall From Grace

- Preprocessor adds concision, expressivity, and abstraction
  - Conditional compilation
    - Code has many meanings
  - Macros
    - Code doesn’t mean what’s written
- File inclusion
The Fall From Grace

- Preprocessor adds concision, expressivity, and abstraction
  - Conditional compilation
    - Code has many meanings
  - Macros
    - Code doesn’t mean what’s written
- File inclusion
  - Code is incomplete
The Fall From Grace

- Preprocessor adds concision, expressivity, and abstraction
  - Conditional compilation
    - Code has many meanings
  - Macros
    - Code doesn’t mean what’s written
- File inclusion
  - Code is incomplete
- Preprocessor only works on tokens, not C constructs
Back to Eden?

• One configuration at a time (preprocess first)
  • Causes exponential explosion
• Incomplete heuristic parsing algorithm
  • Works, *if* you don’t write the wrong code
• Plug-in architecture for further cpp idioms
  • Creates arms race: tool builders vs developers
More of the Same

- Adams et. al, AOSD ’09
- Akers et. al, WCRE ’05
- Badros and Notkin, SP&E ’00
- Baxter and Mehlich, WCRE ’01
- Favre, IWPC ’97
- Garrido and Johnson, ICSM ’05
- McCloskey and Brewer, ESEC ’05
- Padioleau, CC ’09
- Spinellis, TSE ’03
- Vittek, CSMR ’03
Not Quite the Same

- MAPR by Platoff et al., ICSM ’91
- TypeChef by Kästner et al., OOPSLA ’11
This Talk

• Introduction

• **Problem and Solution Approach**

• The Configuration-Preserving Preprocessor

• The Configuration-Preserving Parser

• Our Tool and Evaluation

• Conclusion
#include "eden.h"

cchar eve(void) {
    #ifndef FEAR_OF_GOD
        eat(fruit);
        gain(wisdom);
    #endif
        return cain + abel;
}
```c
#define gain(x) goto banishment

#include "eden.h"

cchar eve(void) {
  #ifndef FEAR_OF_GOD
    eat(fruit);
    gain(wisdom);
  #endif
    return cain + abel;
}
```
#define gain(x) goto banishment

char eve(void) {
  ifndef FEAR_OF_GOD
    eat(fruit);
    gain(wisdom);
  endif
    return cain + abel;
}
char eve(void) {
    eat(fruit);
    goto banishment;
    return cain + abel;
}
char eve(void) {
    #ifndef FEAR_OF_GOD
        eat(fruit);
        goto banishment;
    #endif
    return cain + abel;
}

char eve(void) {
  #ifndef FEAR_OF_GOD
    eat(fruit);
    goto banishment;
  #endif
  return cain + abel;
}

Treat as free: neither defined nor undefined
```c
char eve(void) {
    #ifndef FEAR_OF_GOD
        eat(fruit);
        goto banishment;
    #endif
    return cain + abel;
}
```

Treat as *free*: neither defined nor undefined

Fork parser state on conditional
char eve(void) {
    #ifndef FEAR_OF_GOD
        eat(fruit);
        goto banishment;
    #endif
    return cain + abel;
}

Treat as \textit{free}: neither defined nor undefined

Fork parser state on conditional

Merge parser states again after conditional
char
eve(void)

Function Definition

Compound Statement

Static Choice

! FEAR_OF_GOD

Expression Statement
eat(fruit)

Goto Statement
banishment

Return Statement
cain + abel
Solution Approach

1. Lex to produce tokens (as before)
2. Preprocess while preserving conditionals
3. Parse across conditionals
The first step is lexing. The lexer converts raw program text into tokens, stripping layout such as whitespace and comments. Layout annotations are essential to enable the parser to access the target construct. The second step is preprocessing. It collects macro definitions (Macro (un)definitions).

Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactoring—besides annotating tokens with layout and with preprocessor needs to be configured with the ground truth of layout. SuperC for now.) Additionally, since C compilers have built-in object-like macros, such as originally written

The preprocessor records definitions and undefinitions for the same macro may appear as a macro, defined macro that depends on the configuration. Figure 3 shows

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

The preprocessor records definitions and undefinitions, so that it can determine which macros are neither defined nor undefined and thus also removing infeasible entries on each update. The preprocessor super preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

wherever multiply-defined macros are used, they propagate their definitions in di

A multiply-defined macro from include

The second step is preprocessing. It collects macro definitions (Macro (un)definitions).

Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactoring—besides annotating tokens with layout and with preprocessor needs to be configured with the ground truth of layout. SuperC for now.) Additionally, since C compilers have built-in object-like macros, such as originally written

The preprocessor records definitions and undefinitions for the same macro may appear as a macro, defined macro that depends on the configuration. Figure 3 shows

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

The preprocessor records definitions and undefinitions, so that it can determine which macros are neither defined nor undefined and thus also removing infeasible entries on each update. The preprocessor super preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

wherever multiply-defined macros are used, they propagate their definitions in di

A multiply-defined macro from include

The second step is preprocessing. It collects macro definitions (Macro (un)definitions).

Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactoring—besides annotating tokens with layout and with preprocessor needs to be configured with the ground truth of layout. SuperC for now.) Additionally, since C compilers have built-in object-like macros, such as originally written

The preprocessor records definitions and undefinitions for the same macro may appear as a macro, defined macro that depends on the configuration. Figure 3 shows

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

The preprocessor records definitions and undefinitions, so that it can determine which macros are neither defined nor undefined and thus also removing infeasible entries on each update. The preprocessor super preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

wherever multiply-defined macros are used, they propagate their definitions in di

A multiply-defined macro from include

The second step is preprocessing. It collects macro definitions (Macro (un)definitions).

Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactoring—besides annotating tokens with layout and with preprocessor needs to be configured with the ground truth of layout. SuperC for now.) Additionally, since C compilers have built-in object-like macros, such as originally written

The preprocessor records definitions and undefinitions for the same macro may appear as a macro, defined macro that depends on the configuration. Figure 3 shows

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

The preprocessor records definitions and undefinitions, so that it can determine which macros are neither defined nor undefined and thus also removing infeasible entries on each update. The preprocessor super preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

wherever multiply-defined macros are used, they propagate their definitions in di

A multiply-defined macro from include

The second step is preprocessing. It collects macro definitions (Macro (un)definitions).

Since lexing is performed before preprocessing and parsing, it does not interact with the other two steps. However, automated refactoring—besides annotating tokens with layout and with preprocessor needs to be configured with the ground truth of layout. SuperC for now.) Additionally, since C compilers have built-in object-like macros, such as originally written

The preprocessor records definitions and undefinitions for the same macro may appear as a macro, defined macro that depends on the configuration. Figure 3 shows

Macro invocations. Since macros may be nested within each other, a configuration-preserving preprocessor, just like an ordinary preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

The preprocessor records definitions and undefinitions, so that it can determine which macros are neither defined nor undefined and thus also removing infeasible entries on each update. The preprocessor super preprocessor, needs to recursively expand each macro. Furthermore, since C compilers have built-in object-like macros, such a macro, defined macro

wherever multiply-defined macros are used, they propagate their definitions in di
Three Key Techniques

- **Track presence conditions**
  - **In preprocessor and parser**
- **Hoist conditionals**
  - **In preprocessor**
- **Contain number of forked subparsers**
  - **In parser**
This Talk

- Introduction
- Problem and Solution Approach
- The Configuration-Preserving Preprocessor
- The Configuration-Preserving Parser
- Our Tool and Evaluation
- Conclusion
Track Presence Conditions

- Do not evaluate conditional expressions
- Rather translate to boolean functions (BDDs)
  - Enables reasoning about configurations
    - Including equivalent and invalid branches
- Four base cases
  - Constants, free macros, defined operator
    - Straight-forward case analysis
  - Arithmetic expressions
    - Treat as opaque: no known efficient algorithm
Track Presence Conditions

- Maintain conditional and ordered macro table
  - Contains all definitions of a macro
    - Invocation is an implicit conditional

```c
#if defined GENESIS
#define ARK noah
#elif defined EXODUS
#define ARK the_covenant
#endif
```
Track Presence Conditions

- Maintain conditional and ordered macro table
- Contains all definitions of a macro
- Invocation is an implicit conditional

#if defined GENESIS
#define ARK noah
#elif defined EXODUS
#define ARK the_covenant
#endif
Hoist Conditionals

• Preprocessor invocations do not compose
  • Directives are exactly one source line
  • Operators only work on tokens

```c
float ark_of_ ##
#if defined GENESIS
   noah;
#else if defined EXODUS
   the_covenant;
#endif
```
Hoist Conditionals

- Preprocessor invocations do not compose
  - Directives are exactly one source line
  - Operators only work on tokens

```c
float ark_of_ ##
#if defined GENESIS
  noah;
#elif defined EXODUS
  the_covenant;
#endif
```

```c
float
#if defined GENESIS
  ark_of_ ## noah;
#elif defined EXODUS
  ark_of_ ## the_covenant;
#endif
```
Algorithm 1 Hoisting Conditionals

1: \textbf{procedure} HOIST\(c, \tau\)
2: \hphantom{1:}▷ Initialize a new conditional with an empty branch.
3: \hphantom{1:}\hspace{1em}C \leftarrow \ [(c, \bullet)]
4: \hphantom{1:}\hspace{1em}\textbf{for all} \ a \in \tau \ \textbf{do}
5: \hphantom{1:}\hspace{2em}if \ a \text{ is a language token} \textbf{then}
6: \hphantom{1:}\hspace{3em}▷ Append \ a \text{ to all branches in} \ C.
7: \hphantom{1:}\hspace{3em}C \leftarrow \ [(c_i, \tau_i a) \mid (c_i, \tau_i) \in C]
8: \hphantom{1:}\hspace{2em}\textbf{else} \hspace{1em}▷ \ a \text{ is a conditional.}
9: \hphantom{1:}\hspace{3em}▷ \text{Recursively hoist conditionals in each branch.}
10: \hphantom{1:}\hspace{2em}B \leftarrow \ [b \mid b \in \text{HOIST}(c_i, \tau_i) \text{ and } (c_i, \tau_i) \in a]
11: \hphantom{1:}\hspace{2em}▷ \text{Combine with already hoisted conditionals.}
12: \hphantom{1:}\hspace{2em}C \leftarrow C \times B
13: \hphantom{1:}\hspace{1em}\textbf{end if}
14: \hphantom{1:}\hspace{1em}\textbf{end for}
15: \hphantom{1:}\hspace{1em}\textbf{return} \ C
16: \hphantom{1:}\hspace{1em}\textbf{end procedure}
Algorithm 1 Hoisting Conditionals

1: procedure Hoist(C, \( \tau \))
2: \( \triangleright \) Initialize a new conditional with an empty branch.
3: \( C \leftarrow [ (c, \bullet) ] \)
4: for all \( a \in \tau \) do
5: \( \quad \) if \( a \) is a language token then
6: \( \quad \quad \triangleright \) Append \( a \) to all branches in \( C \)
7: \( \quad \quad C \leftarrow [ (c, \tau_i a) \mid (c, \tau_i) \in C ] \)
8: \( \quad \) else \( \triangleright \) \( a \) is a conditional.
9: \( \quad \quad \triangleright \) Recursively hoist conditionals in each branch.
10: \( \quad \quad \triangleright \) Combine with already hoisted conditionals.
11: \( \quad \) end if
12: \( \quad \) end for
13: \( \) end if
14: \( \) end for
15: \( \) return \( C \)
16: \( \) end procedure

**Hoist Conditionals**

- Ensure that tokens appear only in innermost conditional branches
- If necessary, duplicate tokens across branches
- Special version for function-like macros
- Parse invocation and hoist at same time
This Talk

• Introduction
• Problem and Solution Approach
• The Configuration-Preserving Preprocessor
• The Configuration-Preserving Parser
• Our Tool and Evaluation
• Conclusion
Review: LR Parser

Finite Control

Input

Stack
Review: LR Parser

Finite Control

Shift token from input to stack

Input

Stack
Review: LR

Finite Control

Shift token from input to stack

Reduce top $n$ elements to nonterminal

Stack

Input
Finite Control

Use table for actions and transitions

Reduce top $n$ elements to nonterminal

Shift token from input to stack

Input

Stack
Why Build on LR?

• Has explicit state and stack
  • Straight-forward to fork and merge
• Is table-driven
  • Can make it fast
• Supports left-recursion
  • But shift-reduce and reduce-reduce conflicts
Fork-Merge LR (FMLR)

• Input: Preprocessed source code
  • Ordinary tokens from C language

• Main data structure: Subparser
  • Head, i.e., next token
  • LR stack
Fork-Merge LR (FMLR)

- Input: Preprocessed source code
  - Ordinary tokens from C language
  - Entire conditionals
    - Branches contain tokens and conditionals
- Main data structure: Subparser
  - Presence condition
  - Head, i.e., next token or conditional
  - LR stack
Algorithm 2 Fork-Merge LR Parsing

1: **procedure** PARSE($a_0$)  
2: $Q$.init((true, $a_0$, $s_0$))  \(\triangleright\) The initial subparser for $a_0$.  
3: **while** $Q \neq \emptyset$ **do**  
4: \hspace{1em} $p \leftarrow Q$.pull()  \(\triangleright\) Step the next subparser.  
5: \hspace{1em} $T \leftarrow$ FOLLOW($p.c$, $p.a$)  
6: \hspace{1em} **if** $|T| = 1$ **then**  
7: \hspace{2em} \(\triangleright\) Do an LR action and reschedule the subparser.  
8: \hspace{2em} $Q$.insert(LR($T(1)$, $p$))  
9: \hspace{2em} **else** \(\triangleright\) The follow-set contains several tokens.  
10: \hspace{2em} \(\triangleright\) Fork subparsers and reschedule them.  
11: \hspace{2em} $Q$.insertAll(FORK($T$, $p$))  
12: \hspace{1em} **end if**  
13: \hspace{1em} $Q \leftarrow$ MERGE($Q$)  
14: **end while**  
15: **end procedure**
Algorithm 2 Fork-Merge LR Parsing

1: procedure \textsc{Parse}(a_0)
2: \hspace{1em} Q.init((true, a_0, s_0)) \triangleright The initial subparser for \( a_0 \).
3: \hspace{1em} while \( Q \neq \emptyset \) do
4: \hspace{2em} \( p \leftarrow Q.\text{pull()} \) \triangleright Step the next subparser.
5: \hspace{2em} \( T \leftarrow \text{Follow}(p.c, p.a) \)
6: \hspace{2em} if \( |T| = 1 \) then
7: \hspace{3em} \triangleright Do an LR action and reschedule the subparser.
8: \hspace{3em} \( Q.\text{insert}((\text{LR}(T(1), p)) \)
9: \hspace{3em} else \triangleright The follow-set contains several tokens.
10: \hspace{3em} \triangleright Fork subparsers and reschedule them.
11: \hspace{3em} \( Q.\text{insertAll}(\text{Fork}(T, p)) \)
12: \hspace{2em} end if
13: \hspace{1em} \( Q \leftarrow \text{Merge}(Q) \)
14: \hspace{1em} end while
15: end procedure
Algorithm 2 Fork-Merge LR Parsing

1: procedure Parse($a_0$)  
2:     $Q$.init(((true, $a_0$, $s_0$)) ▷ The initial subparser for $a_0$.  
3:     while $Q \neq \emptyset$ do  
4:         $p \leftarrow Q$.pull() ▷ Step the next subparser.  
5:         $T \leftarrow$ FOLLOW($p.c$, $p.a$)  
6:         if $|T| = 1$ then  
7:             ▷ Do an LR action and reschedule the subparser.  
8:                 $Q$.insert(LR($T(1)$, $p$))  
9:         else ▷ The follow-set contains several tokens.  
10:            ▷ Fork subparsers and reschedule them.  
11:                 $Q$.insertAll(FORK($T$, $p$))  
12:         end if  
13:     end while  
14: end procedure
Algorithm 2 Fork-Merge LR Parsing

1: procedure PARSE(a₀)
2:     Q.init((true, a₀, s₀))▷ The initial subparser for a₀.
3: while Q ≠ ∅ do
4:     p ← Q.pull()▷ Step the next subparser.
5:     T ← FOLLOW(p.c, p.a)
6:     if |T| = 1 then
7:        ▷ Do an LR action and reschedule the subparser.
8:         Q.insert(LR(T(1), p))
9:     else▷ The follow-set contains several tokens.
10:       ▷ Fork subparsers and reschedule them.
11:         Q.insertAll(FORK(T, p))
12:     end if
13:     Q ← MERGE(Q)
14: end while
15: end procedure
Algorithm 2 Fork-Merge LR Parsing

1: procedure Parse($a_0$)
2: $Q$.init((true, $a_0$, $s_0$)) // The initial subparser for $a_0$.
3: while $Q \neq \emptyset$ do
4:     $p \leftarrow Q$.pull() // Step the next subparser.
5:     $T \leftarrow$ Follow($p.c$, $p.a$)
6:     if $|T| = 1$ then
7:         // Do an LR action and reschedule the subparser.
8:         $Q$.insert(LR($T(1)$, $p$))
9:     else // The follow-set contains several tokens.
10:        // Fork subparsers and reschedule them.
11:        $Q$.insertAll(Fork($T$, $p$))
12:    end if
13: $Q \leftarrow$ Merge($Q$)
14: end while
15: end procedure
**Algorithm 2 Fork-Merge LR Parsing**

1: **procedure** `PARSE(a_0)`
2: \[Q\text{.init}((\text{true}, a_0, s_0))\] \[\triangleright\] The initial subparser for \(a_0\).
3: \[\textbf{while } Q \neq \emptyset \textbf{ do}\]
4: \[p \leftarrow Q\text{.pull}()\] \[\triangleright\] Step the next subparser.
5: \[T \leftarrow \text{FOLLOW}(p.c, p.a)\]
6: \[\textbf{if } |T| = 1 \textbf{ then}\]
7: \[\triangleright\] Do an LR action and reschedule the subparser.
8: \[Q\text{.insert}(\text{LR}(T(1), p))\]
9: \[\textbf{else }\] \[\triangleright\] The follow-set contains several tokens.
10: \[\triangleright\] Fork subparsers and reschedule them.
11: \[Q\text{.insertAll}(\text{FORK}(T, p))\]
12: \[\textbf{end if}\]
13: \[Q \leftarrow \text{MERGE}(Q)\]
14: \[\textbf{end while}\]
15: **end procedure**
Algorithm 2 Fork-Merge LR Parsing

1: procedure PARSE($a_0$)
2:     $Q$.init((true, $a_0$, $s_0$)) ▷ The initial subparser for $a_0$.
3:     while $Q \neq \emptyset$ do
4:         $p \leftarrow Q$.pull() ▷ Step the next subparser.
5:         $T \leftarrow \text{FOLLOW}(p.c, p.a)$
6:         if $|T| = 1$ then
7:             ▷ Do an LR action and reschedule the subparser.
8:             $Q$.insert(LR($T(1)$, $p$))
9:         else ▷ The follow-set contains several tokens.
10:            $Q$.insertAll(FORK($T$, $p$))
11:        end if
12:    end while
13: end procedure
Algorithm 2 Fork-Merge LR Parsing

1: procedure \textsc{Parse}(a_0)  
2: \hspace{1em} \text{Q.init}((true, a_0, s_0)) \triangleright \text{The initial subparser for } a_0.  
3: \hspace{1em} \textbf{while } Q \neq \emptyset \textbf{ do}  
4: \hspace{2em} p \leftarrow Q\text{.pull()} \triangleright \text{Step the next subparser.}  
5: \hspace{2em} T \leftarrow \text{Follow}(p.c, p.a)  
6: \hspace{2em} \textbf{if } |T| = 1 \textbf{ then}  
7: \hspace{3em} \triangleright \text{Do an LR action and reschedule the subparser.}  
8: \hspace{3em} Q\text{.insert}((LR(T(1), p))  
9: \hspace{2em} \textbf{else} \triangleright \text{The follow-set contains several tokens.}  
10: \hspace{3em} \triangleright \text{Fork subparsers and reschedule them.}  
11: \hspace{3em} Q\text{.insertAll}(\text{Fork}(T, p))  
12: \hspace{2em} \textbf{end if}  
13: \hspace{1em} Q \leftarrow \textsc{Merge}(Q)  
14: \hspace{1em} \textbf{end while}  
15: \hspace{1em} \textbf{end procedure}
Fork-Merge LR Highlights

• Fork subparsers from other subparsers
  • Have different heads, presence conditions
  • But reuse LR stack frames in a DAG
Fork-Merge LR Highlights

• Fork subparsers from other subparsers
  • Have different heads, presence conditions
  • But reuse LR stack frames in a DAG
• Merge subparsers if same heads and stacks
  • Disjoin presence conditions
  • Priority queue ensures merging happens ASAP
Fork-Merge LR Highlights

- Fork subparsers from other subparsers
  - Have different heads, presence conditions
  - But reuse LR stack frames in a DAG
- Merge subparsers if same heads and stacks
  - Disjoin presence conditions
  - Priority queue ensures merging happens ASAP
- Perform ordinary LR on ordinary tokens
  - Reuse grammars and table generators
The Merge Criterion

- Merge only if same heads and stacks
  - Implies same derivation of nonterminals
    - Which also ensures a well-formed AST
  - But may parse some code more than once

```c
#define GENESIS
  if (boarded(ark))
    survive();
  else
#define endif
  drown();
```
The Merge Criterion

- Merge only if same heads and stacks
  - Implies same derivation of nonterminals
    - Which also ensures a well-formed AST
  - But may parse some code more than once

```c
#ifdef GENESIS
  if (boarded(ark))
    survive();
  else
#endif
  drown();
Parese twice and merge thereafter
```
The Merge Criterion

- Merge only if same heads and stacks
- Implies same derivation of nonterminals
- Which also ensures a well-formed AST
- But may parse some code more than once

```c
#ifdef GENESIS
if (boarded(ark))
    survive();
else
#endif
    drown();
```
The Serpent Is in the Details

Which subparsers to fork?
Which Subparsers to Fork?

- Naively, subparser for each conditional branch
  - But conditionals may
    - Be directly nested in each other
    - Directly follow each other
  - Creates many unnecessary subparsers!
Which Subparsers to Fork?

• Insight: Real (LR) work performed on tokens
  • Therefore, create subparser for each token
    • Directly reachable from current position
  • Across all configurations
#ifndef EDEN
#endif

#ifdef THE_FALL
    cherubim(); fiery_sword();
#endif
#endif

#ifdef BABEL
    languages(4294967295);
#endif
#endif

sense_of_loss();
#ifdef EDEN
    #ifdef THE_FALL
        cherubim(); fiery_sword();
    #endif
#endif

    #ifdef BABEL
        languages(4294967295);
    #endif

sense_of_loss();
```c
#ifdef EDEN
#ifdef THE_FALL
    cherubim(); fiery_sword();
#endif
#endif

#ifdef BABEL
    languages(4294967295);
#endif

sense_of_loss();
```
#ifdef EDEN
	#ifdef THE_FALL
		cherubim(); fiery_sword();
	#endif
#endif

#ifdef BABEL
	languages(4294967295);
#endif
	sense_of_loss();
#ifdef EDEN
    #ifdef THE_FALL
        cherubim(); fiery_sword();
    #endif
#endif

#ifdef BABEL
    languages(4294967295);
#endif

sense_of_loss();
```c
#ifdef EDEN
#ifdef THE_FALL
    cherubim(); fiery_sword();
#endif
#endif
#ifdef BABEL
    languages(4294967295);
#endif
sense_of_loss();
```

Algorithm 3 The Token Follow-Set

1: procedure Follow(c, a)
2: \[ T \leftarrow \emptyset \] \quad \triangleright \text{Initialize the follow-set.}
3: procedure First(c, a)
4: \quad loop
5: \quad \quad if \ a \text{ is a language token} \ then
6: \quad \quad \quad T \leftarrow T \cup \{(c, a)\}
7: \quad \quad return false
8: \quad else \quad \quad \triangleright \ a \text{ is a conditional.}
9: \quad \quad c_r \leftarrow false \quad \triangleright \text{Initialize remaining condition.}
10: \quad \quad for all (c_i, \tau_i) \in a \ do
11: \quad \quad \quad if \ \tau_i = \circ \ then
12: \quad \quad \quad \quad c_r \leftarrow c_r \lor c \land c_i
13: \quad \quad \quad else
14: \quad \quad \quad \quad c_r \leftarrow c_r \lor \text{First}(c \land c_i, \tau_i(1))
15: \quad \quad \quad end if
16: \quad \quad end for
17: \quad \quad if \ c_r = false \ or \ a \text{ is last element in branch} \ then
18: \quad \quad \quad return c_r
19: \quad \quad end if
20: \quad \quad c \leftarrow c_r
21: \quad \quad a \leftarrow \text{next token or conditional after } a
22: \quad \quad end if
23: \quad end loop
24: end procedure
25: loop
26: \quad c \leftarrow \text{First}(c, a)
27: \quad if \ c = false \ then \text{return } T \ end if \quad \triangleright \text{Done.}
28: \quad a \leftarrow \text{next token or conditional after } a
29: end loop
30: end procedure
The Token Follow-Set

- Iterate over consecutive conditionals until reaching a token for every configuration
- Recurse over conditional branches
- Track presence condition
  - For remaining configuration
  - For each token

---

**Algorithm 3 The Token Follow-Set**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>procedure FOLLOW(c, a)</td>
</tr>
<tr>
<td>2:</td>
<td>$T \leftarrow \emptyset$ &gt; Initialize the follow-set.</td>
</tr>
<tr>
<td>3:</td>
<td>procedure FIRST(c, a)</td>
</tr>
<tr>
<td>4:</td>
<td>$T \leftarrow \emptyset$ &gt; Language token field</td>
</tr>
<tr>
<td>5:</td>
<td>$T \leftarrow T \cup {(c, a)}$</td>
</tr>
<tr>
<td>6:</td>
<td>return false</td>
</tr>
<tr>
<td>7:</td>
<td>else $a$ is a conditional.</td>
</tr>
<tr>
<td>8:</td>
<td>$c_r \leftarrow false$ &gt; Initialize remaining condition.</td>
</tr>
<tr>
<td>9:</td>
<td>for all $(c_i, \tau_i) \in a$ do</td>
</tr>
<tr>
<td>10:</td>
<td>for all $(c_i, \tau_i) \in a$ do</td>
</tr>
<tr>
<td>11:</td>
<td>$c_r \leftarrow c_r \vee c_i \wedge c_i$</td>
</tr>
<tr>
<td>12:</td>
<td>$c_r \leftarrow c_r \vee FIRST(c_i, \tau_i(1))$</td>
</tr>
<tr>
<td>13:</td>
<td>end if</td>
</tr>
<tr>
<td>14:</td>
<td>end if</td>
</tr>
<tr>
<td>15:</td>
<td>if $c_r = false$ or $a$ is last element in branch then</td>
</tr>
<tr>
<td>16:</td>
<td>return $c_r$</td>
</tr>
<tr>
<td>17:</td>
<td>end if</td>
</tr>
<tr>
<td>18:</td>
<td>$a \leftarrow$ next token or conditional after $a$</td>
</tr>
<tr>
<td>19:</td>
<td>end if</td>
</tr>
<tr>
<td>20:</td>
<td>end procedure</td>
</tr>
<tr>
<td>21:</td>
<td>loop</td>
</tr>
<tr>
<td>22:</td>
<td>$T = false$ then return $T$ end if &gt; Done.</td>
</tr>
<tr>
<td>23:</td>
<td>$a \leftarrow$ next token or conditional after $a$</td>
</tr>
<tr>
<td>24:</td>
<td>end loop</td>
</tr>
<tr>
<td>25:</td>
<td>end procedure</td>
</tr>
</tbody>
</table>
The Serpent Is in the Details

There is more!
const char * versions[] = {
#ifdef KING_JAMES
    "and, behold, it was very good",
#endif
#ifdef NEW_INTERNATIONAL
    "and it was very good",
#endif
#ifdef
    // And so on...
    NULL
};
Encodes binary number!

```c
const char * versions[] = {
#ifdef KING_JAMES
    "and, behold, it was very good",
#endif
#ifdef NEW_INTERNATIONAL
    "and it was very good",
#endif
    // And so on...
    NULL
};
```
const char * versions[] = {
#ifdef KING_JAMES
    "and, behold, it was very good",
#endif
#ifdef NEW_INTERNATIONAL
    "and it was very good",
#endif
    // And so on...
    NULL
};
const char * versions[] = {
#ifdef KING_JAMES
    "and, behold, it was very good",
#endif
#ifdef NEW_INTERNATIONAL
    "and it was very good",
#endif
    // And so on...
    NULL
};

Shared Reduces: Reduce same stack for several heads
const char * versions[] = {
#ifdef KING_JAMES
   "and, behold, it was very
#endif
#ifdef NEW_INTERNATIONAL
   "and it was very good",
#endif
   // And so on...
   NULL
};

Shared Reduces: Reduce same stack for several heads

Lazy Shifts: Delay forking for several heads
const char * versions[] = {
    #ifdef KING_JAMES
        "and, behold, it was very
    #endif
    #ifdef NEW_INTERNATIONAL
        "and it was very good",
    #endif
    // And so on...
    NULL
};

Shared Reduces:
Reduce same stack for several heads

Lazy Shifts:
Delay forking for several heads

Early Reduces:
Pick reducing before shifting subparser
const char * versions[] = {
    #ifdef KING_JAMES
        "and, behold, it was very good",
    #endif
    #ifdef NEW_INTERATIONAL
        "and it was very good",
    #endif
    // And so on...
    NULL
};

Uses only two subparsers!
Shared Reduces & Lazy Shifts

• Require *multi-headed* subparser
  • Several tokens and their presence conditions
  • But only one LR stack
• Prioritized by first head
• Fork by removing one head, presence condition
• Merge by disjoining presence conditions
  • For same heads, LR stack
This Talk

• Introduction
• Problem and Solution Approach
• The Configuration-Preserving Preprocessor
• The Configuration-Preserving Parser
• Our Tool and Evaluation
• Conclusion
<table>
<thead>
<tr>
<th><strong>SuperC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td><strong>Preprocessor</strong></td>
</tr>
<tr>
<td><strong>Parser formalism</strong></td>
</tr>
<tr>
<td><strong>Grammar</strong></td>
</tr>
<tr>
<td><strong>Forking &amp; Merging</strong></td>
</tr>
<tr>
<td><strong>AST</strong></td>
</tr>
<tr>
<td><strong>Presence conditions</strong></td>
</tr>
<tr>
<td><strong>Complete</strong></td>
</tr>
<tr>
<td>Feature</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td><strong>Preprocessor</strong></td>
</tr>
<tr>
<td><strong>Parser formalism</strong></td>
</tr>
<tr>
<td><strong>Grammar</strong></td>
</tr>
<tr>
<td><strong>Forking &amp; Merging</strong></td>
</tr>
<tr>
<td><strong>AST</strong></td>
</tr>
<tr>
<td><strong>Presence conditions</strong></td>
</tr>
<tr>
<td><strong>Complete</strong></td>
</tr>
</tbody>
</table>
And MAPR?

- Preprocessor is not documented
- Parser boils down to naive FMLR
  - Fork on every conditional branch
  - No other optimizations
- Reimplemented in SuperC as option
Who Is Our Serpent?
Who Is Our Serpent?

Linux!
The Linux OS

- Critical to government and business
- Jarring flexibility and performance requirements
  - From smart phones to cloud computing farms
- Large and complex
- Many developers with varying styles, skills
Evaluation Plan

• Focus on x86 Linux kernel
  • How complex is preprocessor usage?
    • Compare to MAPR along the way
  • How well does SuperC perform?
    • Compare to TypeChef along the way
Linux Preprocessor Usage

• Breadth of conditionals
  • Force forking of parser states

• Incidence of incomplete C constructs
  • Prevent merging of parser states
Linux Preprocessor Usage

• Breadth of conditionals
  • Force forking of parser states
• Incidence of incomplete C constructs
  • Prevent merging of parser states
• Count subparsers per FMLR loop iteration!
Cumulative Distribution of Subparser # per Iteration

![Graph showing cumulative distribution of subparser numbers per iteration.]

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Number of Subparsers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>21</td>
</tr>
<tr>
<td>0.50</td>
<td>77</td>
</tr>
<tr>
<td>0.75</td>
<td>39</td>
</tr>
<tr>
<td>1.00</td>
<td>468</td>
</tr>
</tbody>
</table>

- **Shared**: 21 (99%) to 77 (100%)
- **All**: 21 (99%) to 39 (100%)
- **Follow-Set**: 33 (99%) to 468 (100%)

**MAPR**: >16,000 on 98% of C files
Cumulative Distribution of Latency per C File

Max: 10.4
SuperC

Max: 931
TypeChef

Fraction

0.25
0.50
0.75
1.00

Latency in Seconds

0 5 10 15 20 25 30 35 40
• Totals, averages, and medians are deceiving
• We really need the entire distribution!
Latency vs File Size

Parse, Preprocess, Lex

Latency in Seconds vs Size in Millions of Bytes
Future Work

• Analyze names and types across conditionals
  • Foundation for all language processing tools
• Annotate AST with layout, macros, includes
  • Refactorings need to print source code again
    • Modulo intended changes
Conclusions

• Preprocessor makes it hard to parse all of C
  • No viable solution for 40 years!
• We performed a systematic analysis of problem
  • Identified solution strategies
    • Notably: Conditional hoisting, FMLR parsing
  • And implemented a real-world tool, SuperC
• On x86 Linux kernel, SuperC is fast and scales well
SuperC

http://cs.nyu.edu/xtc/
SuperC Context Management

• C syntax is context-sensitive: \( T \times p ; \)
  • If \( T \) is a type name, declare \( p \) as a pointer
  • Otherwise, calculate the product of \( T \) and \( p \)
• Worse, C with conditionals can be ambiguous
  • Same name may be both a type and an object
  • Under different presence conditions
SuperC
Context Management

• C syntax is context-sensitive: $T \ast p$;
  • If $T$ is a type name, declare $p$ as a pointer
  • Otherwise, calculate the product of $T$ and $p$
• Worse, C with conditionals can be ambiguous
  • Same name may be both a type and an object
    • Under different presence conditions
• SuperC needs to fork and merge subparsers!
SuperC
Context Management

- Provide context object for each subparser

- Customize forking and merging
  - Reclassify
  - Fork context
  - May merge
  - Merge contexts
SuperC
Context Management

- Provide context object for each subparser
  - Conditional, scoped symbol table
- Customize forking and merging
  - Reclassify: check for ambiguously defined name
  - Fork context: duplicate symbol table reference
  - May merge: check number of nested scopes
  - Merge contexts: combine symbol table entries