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
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, Mac OS, 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 More of the Same

• TypeChef by Kästner et al., OOPSLA ’11
This Talk

- Introduction
- Problem and Solution Approach
- The Core Algorithm
- The Real World
- Conclusion
```c
#include "eden.h"

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

#include "eden.h"

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

Treat as free: neither defined nor undefined
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;
}
Solution Approach

1. Lex to produce tokens (as before)

2. Preprocess while preserving conditionals
   - Treat configuration variables as free
     - Unless set with `#define` and `#undefine`

3. Parse across conditionals
   - Split parser states on conditionals
   - Merge parser states after conditionals
### Preprocessor

<table>
<thead>
<tr>
<th>Language &amp; Granularity</th>
<th>Strategy</th>
<th>Surroundings by Conditionals</th>
<th>Contains Conditionals</th>
<th>Spans Conditionals</th>
<th>Contains Multiply-Defined Macros</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Macro (Un)Definition</strong></td>
<td>CPP One line</td>
<td>Use conditional macro table</td>
<td>Multiple entries in macro table</td>
<td></td>
<td>Expansion delayed until invocation</td>
<td>Remove redenitions</td>
</tr>
<tr>
<td><strong>Macro Invocation</strong></td>
<td>CPP Many lines</td>
<td>Expand to all definitions</td>
<td>Expand to valid definitions only</td>
<td>Hoist conditionals around invocation</td>
<td>Hoist conditionals around invocation</td>
<td></td>
</tr>
<tr>
<td><strong>Token Pasting &amp; Stringification</strong></td>
<td>CPP One line</td>
<td>Apply pasting &amp; stringification</td>
<td>Hoist conditional around operations</td>
<td></td>
<td>Hoist conditional around operations</td>
<td></td>
</tr>
<tr>
<td><strong>File Inclusion</strong></td>
<td>CPP One line</td>
<td>Include &amp; preprocess</td>
<td>Preprocess under presence condition</td>
<td></td>
<td>Hoist conditional around inclusion</td>
<td>Reinclude when guard macro is not false</td>
</tr>
<tr>
<td><strong>Conditionals</strong></td>
<td>CPP Many lines</td>
<td>Preprocess tests &amp; all branches</td>
<td>AND nested conditions</td>
<td></td>
<td>Hoist conditionals around expression</td>
<td>Preserve order for non-boolean expressions</td>
</tr>
<tr>
<td><strong>Error Directives</strong></td>
<td>CPP One line</td>
<td>Annotate tokens</td>
<td>Indicates infeasible branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Line, Warning, &amp;Pragma Directives</strong></td>
<td>CPP One line</td>
<td>Annotate tokens</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lexer**

- Annotate tokens

**Parser**

- Fork-merge
- Fork parsers
- Use conditional symbol table
- Fork parsers on ambiguous names
By convention header files protect against multiple inclusion with needs to track implicit presence conditions due to surrounding conflicts over header files. As for macro invocations, the preprocessor resolves file inclusion directives under header files. Line 4 illustrates the token pasting operator. The preprocessor needs to hoist the conditionals outside operator invocations. Line 4 in Figure illustrates the stringification operator. The preprocessor needs to hoist the conditionals outside operator invocations. Consequently, a configuration-preserving preprocessor needs to hoist any multiply-defined macros. Consequently, the invocation on line 4 may be an invocation of macro __cpu_to_le32. Consequently, the invocation on line 4 may be an invocation of macro __cpu_to_le32.

<table>
<thead>
<tr>
<th>Layout</th>
<th>CPP Arbitrary</th>
<th>Annotate tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preprocessor</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Macro (Un)Definition</td>
<td>CPP One line</td>
<td>Use conditional macro table</td>
</tr>
<tr>
<td>Macro Invocation</td>
<td>CPP Many lines</td>
<td>Expand to all definitions</td>
</tr>
<tr>
<td><strong>Token Pasting &amp; Stringification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Token Pasting &amp; Stringification</td>
<td>CPP One line</td>
<td>Include &amp; preprocess</td>
</tr>
<tr>
<td>File Inclusion</td>
<td>CPP Many lines</td>
<td>Preprocess under presence condition</td>
</tr>
<tr>
<td>Conditionals</td>
<td>CPP Many lines</td>
<td>AND nested conditions</td>
</tr>
<tr>
<td>Error Directives</td>
<td>CPP Many lines</td>
<td>Indicates ambiguous names</td>
</tr>
<tr>
<td>Line, Warning, &amp;Pragma Directives</td>
<td>CPP Many lines</td>
<td>Annotate tokens</td>
</tr>
<tr>
<td><strong>Parser</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>CPP Arbitrary</td>
<td>Use conditional symbol table</td>
</tr>
<tr>
<td>Typedef Names</td>
<td>C One token</td>
<td></td>
</tr>
</tbody>
</table>
Three Key Techniques

• Track presence conditions
  • In preprocessor and parser
• Hoist conditionals
  • In preprocessor
• Enforce merge discipline
  • In parser
Track Presence Conditions

- Maintain conjunction of nested conditional tests
- Boolean expressions
  - Combine equivalent branches
  - Trim invalid branches
- Integral expressions
  - Treat as opaque, preserve source code order
Track Presence Conditions

• Maintain conditional and ordered symbol tables
  • Multiply-defined macros
  • Ambiguously-defined C identifiers
    • Typedef names vs object names

#define ARK noah
#else EXODUS
#define ARK the_covenant
#endif
Track Presence Conditions

• Maintain conditional and ordered symbol tables
  • Multiply-defined macros
  • Ambiguously-defined C identifiers
  • Typedef names vs object names

```c
#ifdef GENESIS
#define ARK noah
#elif 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_
#ifdef GENESIS
    noah;
#elif 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_ ##
#ifdef GENESIS
   noah;
#elseif EXODUS
   the_covenant;
#endif
```

```c
float
#ifdef GENESIS
   ark_of_ ## noah;
#elseif EXODUS
   ark_of_ ## the_covenant;
#endif
```
Enforce Merge Discipline

• Forked parsers shall only merge iff
  • They are at the same input position
  • Have the same derivation of nonterminals

```c
#ifdef GENESIS
  if (boarded(ark))
    survive();
  else
#endif
  drown();
```
Enforce Merge Discipline

- Forked parsers shall only merge iff
  - They are at the same input position
  - Have the same derivation of nonterminals

```c
#ifdef GENESIS
    if (boarded(ark))
        survive();
    else
    #endif
    drown();
```
Enforce Merge Discipline

• Forked parsers shall only merge iff
  • They are at the same input position
  • Have the same derivation of nonterminals

```c
#ifdef GENESIS
  if (boarded(ark))
    survive();
  else
#endif
  drown();
```
This Talk

- Introduction
- Problem and Solution Approach
- The Core Algorithm
- The Real World
- Conclusion
Review: LR Parser

Finite Control

Input

Stack
Review: LR Parser

Finite Control

Input

Shift token from input to stack

Stack
Review: LR

Shift token from input to stack

Reduce top $n$ elements to nonterminal

Input

Stack
Review: LR Parser

- Shift token from input to stack
- Reduce top n elements to nonterminal
- Use table for actions and transitions
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 as tokens
  - Regular tokens from C language
  - Compound tokens for `#if`, `#ifdef`, `#ifndef`, ...

Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE(a₀)
2:   \( P \leftarrow \{ \text{initial subparser for } a₀ \} \)
3: loop
4:   for all subparsers \( p \) in \( P \) on the earliest \( p.a \) do
5:     if \( p.a \) is a regular token then
6:       \( \triangleright \) Perform regular LR
7:       LR(\( p \))
8:     else if \( p.a \) is a compound token then
9:       \( \triangleright \) Fork subparsers on the conditional
10:      \( P \leftarrow \text{FORK}(p) \cup (P \setminus p) \)
11:    end if
12: end for
13: if \( P = \emptyset \) then \( \triangleright \) Done
14: return
15: end if
16: \( \triangleright \) Merge subparsers again
17: \( \text{MERGE}(P) \)
18: end loop
19: end procedure

Note: The algorithm describes the procedure for parsing input using the Fork-Merge LR method. It involves iterating over subparsers, performing regular LR actions on regular tokens, handling compound tokens by splitting into subparsers, and finally merging subparsers once all subparsers have been processed, indicating a complete parse. This method allows for efficient parsing of complex grammatical structures while maintaining a compact and readable implementation.
Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE($a_0$)
2:     $P \leftarrow \{ \text{initial subparser for } a_0 \}$
3:     loop
4:         for all subparsers $p$ in $P$ on the earliest $p.a$ do
5:             if $p.a$ is a regular token then
6:                 $\triangleright$ Perform regular LR
7:                 LR($p$)
8:                 else if $p.a$ is a compound token then
9:                     $\triangleright$ Fork subparsers on the conditional
10:                    $P \leftarrow \text{FORK}(p) \cup (P \setminus p)$
11:                     end if
12:             end if
13:         end for
14:     if $P = \emptyset$ then $\triangleright$ Done
15:         return
16:     end if
17:     $\triangleright$ Merge subparsers again
18:     $\text{MERGE}(P)$
19:     end loop

20: end procedure
Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE(a_0)
2:     P ← \{ initial subparser for a_0 \}
3: loop
4:     for all subparsers p in P on the earliest p.a do
5:         if p.a is a regular token then
6:             \( \triangleright \) Perform regular LR
7:             LR(p)
8:         else if p.a is a compound token then
9:             \( \triangleright \) Fork subparsers on the conditional
10:            P ← FORK(p) \( \cup \) (P \( \setminus \) p)
11:        end if
12:     end for
13:     if P = \( \emptyset \) then \( \triangleright \) Done
14:        return
15:     end if
16:     \( \triangleright \) Merge subparsers again
17:     MERGE(P)
18: end loop
19: end procedure
**Algorithm 1** The Fork-Merge LR Parsing Algorithm

1: **procedure** PARSE\((a_0)\)

2: \( P \leftarrow \{\text{initial subparser for } a_0\} \)

3: **loop**

4: \hfill **for all** subparsers \( p \) in \( P \) on the earliest \( p.a \) **do**

5: \hfill **if** \( p.a \) is a regular token **then**

6: \hfill \( \triangleright \) Perform regular LR

7: \hfill \( \text{LR}(p) \)

8: \hfill **else if** \( p.a \) is a compound token **then**

9: \hfill \( \triangleright \) Fork subparsers on the conditional

10: \hfill \( P \leftarrow \text{FORK}(p) \cup (P \setminus p) \)

11: \hfill **end if**

12: **end for**

13: **if** \( P = \emptyset \) **then** \( \triangleright \) Done

14: **return**

15: **end if**

16: \( \triangleright \) Merge subparsers again

17: \( \text{MERGE}(P) \)

18: **end loop**

19: **end procedure**
Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE($a_0$)
2: $P \leftarrow \{\text{initial subparser for } a_0\}$
3: loop
4: for all subparsers $p$ in $P$ on the earliest $p.a$ do
5: if $p.a$ is a regular token then
6: $\triangleright$ Perform regular LR
7: $LR(p)$
8: else if $p.a$ is a compound token then
9: $\triangleright$ Fork subparsers on the conditional
10: $P \leftarrow FORK(p) \cup (P \setminus p)$
11: end if
12: end for
13: if $P = \emptyset$ then $\triangleright$ Done
14: return
15: end if
16: $\triangleright$ Merge subparsers again
17: $MERGE(P)$
18: end loop
19: end procedure
Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE(a₀)
2: P ← \{ initial subparser for a₀ \}
3: loop
4: for all subparsers p in P on the earliest p.a do
5: if p.a is a regular token then
6:▷ Perform regular LR
7: LR(p)
8: else if p.a is a compound token then
9:▷ Fork subparsers on the conditional
10: P ← FORK(p) ∪ (P \ p)
11: end if
12: end for
13: if P = ∅ then▷ Done
14: return
15: end if
16:▷ Merge subparsers again
17: MERGE(P)
18: end loop
19: end procedure
### Algorithm 1 The Fork-Merge LR Parsing Algorithm

1: procedure PARSE($a_0$)
2: \[ P \leftarrow \{ \text{initial subparser for } a_0 \} \]
3: \[ \text{loop} \]
4: \[ \text{for all } \text{subparsers } p \text{ in } P \text{ on the earliest } p.a \text{ do} \]
5: \[ \quad \text{if } p.a \text{ is a regular token then} \]
6: \[ \quad \quad \text{// Perform regular LR} \]
7: \[ \quad \quad \text{LR}(p) \]
8: \[ \quad \quad \text{else if } p.a \text{ is a compound token then} \]
9: \[ \quad \quad \quad \text{// Fork subparsers on the conditional} \]
10: \[ \quad \quad \quad P \leftarrow \text{FORK}(p) \cup (P \setminus p) \]
11: \[ \quad \text{end if} \]
12: \[ \text{end for} \]
13: \[ \quad \text{if } P = \emptyset \text{ then} \quad \text{// Done} \]
14: \[ \quad \text{return} \]
15: \[ \text{end if} \]
16: \[ \quad \text{// Merge subparsers again} \]
17: \[ \quad \text{MERGE}(P) \]
18: \[ \text{end loop} \]
19: \[ \text{end procedure} \]
Fork-Merge LR Highlights

- Perform regular LR on regular tokens
  - Reuse grammars and table generators
- Fork subparsers from other subparsers
  - Reuse LR stack frames in a DAG
- Keep subparsers ordered and merge ASAP
  - Reuse AST fragments as much as possible
The Serpent Is in the Details

• Which subparsers to create on Fork?
  • Intuitively, one for each conditional branch
  • But conditionals may
    • Be directly nested in each other
    • Directly follow each other
  • Therefore, create subparser for each regular token
    • Immediately following current position
    • Across each path through conditionals
Algorithm 2 Compute the set of regular tokens immediately following a compound token starting a conditional.

1: procedure FOLLOW(a)
2:   result ← Ø
3:   ▷ FIRST assumes well-nested conditionals. Its result
4:   ▷ indicates whether to keep on following.
5:   procedure FIRST(a)
6:     loop
7:       if a is a regular token then
8:         result ← {a} ∪ result
9:         return false
10:      else if a ends a branch then
11:         return true
12:      else ▷ a starts a conditional
13:         cont ← false
14:         for all branches b in the conditional do
15:           if FIRST(token after one starting b) then
16:             cont ← true
17:           end if
18:         end for
19:         if conditional has no #else then
20:           cont ← true
21:         end if
22:         if not cont then  ▷ No branch is empty
23:           return false
24:         end if
25:         a ← next token after conditional
26:       end if
27:     end loop
28:   end procedure
29:   loop
30:     if not FIRST(a) then  ▷ Done
31:       return result
32:     end if
33:     ▷ Get next token while stepping out of conditionals
34:     repeat
35:       if a is a regular token then
36:         a ← next token in input
37:       else ▷ a is a compound token
38:         a ← next token after conditional
39:       end if
40:     until a does not end a branch
41:   end loop
42: end procedure
The Follow Set

- Similar to computing follow set for grammars
- Conditional branches instead of alternatives
- Linear scan of input instead of fixed-point
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
   // And so on...
   NULL
};
Encodes binary number!

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
};
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
};

Compute follow set only once

Fork lazily on LR operation
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
};

Compute follow set only once
Fork lazily on LR operation
Merge eagerly after each conditional
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
};

Uses only two subparsers!
This Talk

• Introduction
• Problem and Solution Approach
• The Core Algorithm
• The Real World
• Conclusion
<table>
<thead>
<tr>
<th>Feature</th>
<th>SuperC Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Java</td>
</tr>
<tr>
<td><strong>Preprocessor</strong></td>
<td>Built from scratch</td>
</tr>
<tr>
<td><strong>Parser formalism</strong></td>
<td>FMLR</td>
</tr>
<tr>
<td><strong>Grammar</strong></td>
<td>Modified Roskind’s, ran through Bison</td>
</tr>
<tr>
<td><strong>AST</strong></td>
<td>Grammar annotations</td>
</tr>
<tr>
<td><strong>Typedef names</strong></td>
<td>Per condition</td>
</tr>
<tr>
<td><strong>Presence conditions</strong></td>
<td>BDDs</td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SuperC</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>Java</td>
</tr>
<tr>
<td><strong>Preprocessor</strong></td>
<td>Built from scratch</td>
</tr>
<tr>
<td><strong>Parser formalism</strong></td>
<td>FMLR</td>
</tr>
<tr>
<td><strong>Grammar</strong></td>
<td>Modified Roskind’s, ran through Bison</td>
</tr>
<tr>
<td><strong>AST</strong></td>
<td>Grammar annotations</td>
</tr>
<tr>
<td><strong>Typedef names</strong></td>
<td>Per condition</td>
</tr>
<tr>
<td><strong>Presence conditions</strong></td>
<td>BDDs</td>
</tr>
<tr>
<td><strong>Complete</strong></td>
<td>Yes</td>
</tr>
</tbody>
</table>
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: \( T * p; \)
  • 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
  • Should fork
  • Fork context
  • May merge
  • Merge contexts
SuperC
Context Management

• Provide context object for each subparser
  • Conditional, flat symbol table for current scope only
    • Create on entry, throw away on exit
• Customize forking and merging
  • Should fork: check for ambiguously defined name
  • Fork context: duplicate symbol table reference
• May merge: check number of nested scopes
• Merge contexts: combine symbol table entries
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?
  • 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

- Max 40
- 99th percentile
Cumulative Distribution of Latency per C File

SuperC
Max 14.2

TypeChef
Max 652.2
Sidebar

• Totals, averages, and medians are deceiving
• We really need the entire distribution!
Future Work

- Analyze names and types across conditionals
  - Foundation for all language processing tools
- Annotate AST with layout, macros, inclusions
  - 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
• We developed novel parsing algorithm based on LR
  • And implemented a real-world tool, SuperC
• On x86 Linux kernel, SuperC is fast and scales well
SuperC
SuperC

http://cs.nyu.edu/rgrimm/xtc/