最終レポート

03-133008　ヴーバンタン
03-140427　北里 知也
03-140420　夏　越

1. 達成度
字句解析器，構文解析器，コード生成器のすべてを完成し、 正しい結果が得られた 。

2. 必須課題を越えて行ったことのリスト.
a. 最適化
以下は作ったものの各自のプログラムをうまく合わせることができなかったもの。
b. for
c. コメントアウト
d. 型検査
e. ポインタ（失敗）

3. 速度
各プログラムを作成した cogen と,gcc,gcc -0 3 でコンパイルしたものの実行速度の比較を図 1 に示す。

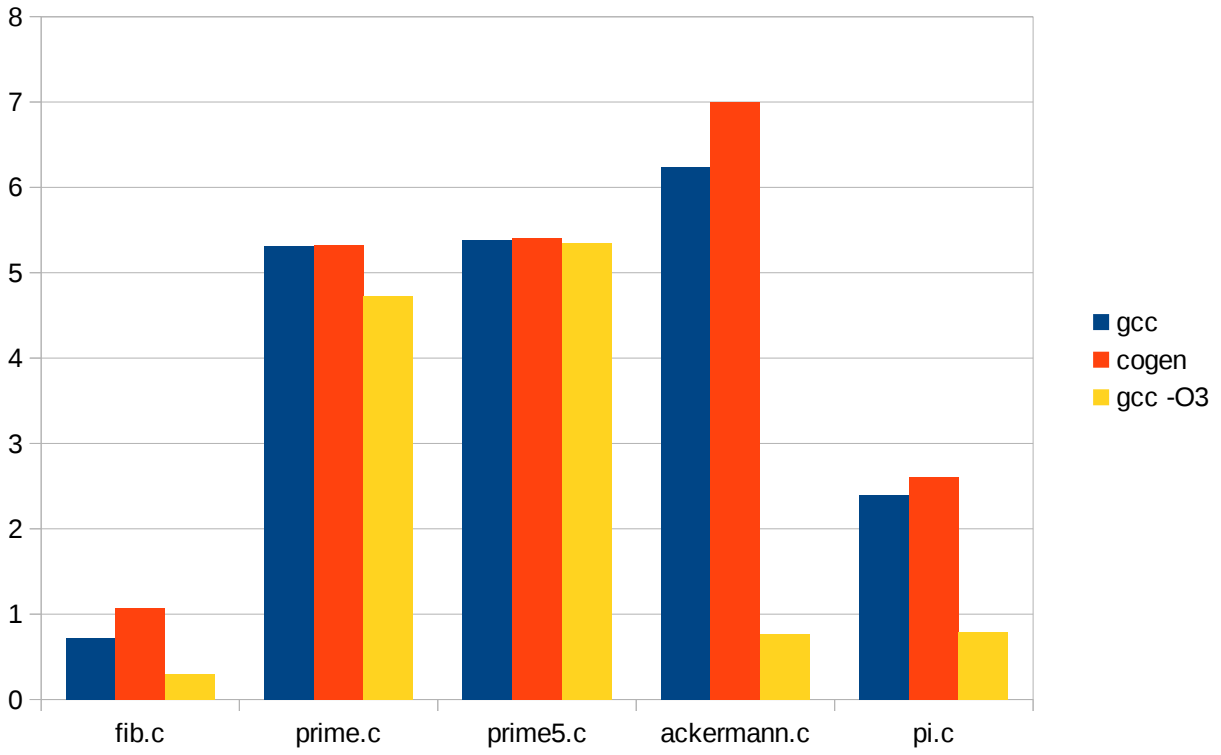

図 1: 実行速度比較

4. コード生成器を自分がどう作ったかに関するまとめ.
演算式に主に利用されるのは三つのレジスタ ebx, esi, edi と避難用のメモリであった。演算式の途中に関数呼び出しが発生する場合、現在の計算値を上記のレジスタがまだ使われていなければそれに保存するが、もし全部が使われていれば避難用のメモリに保存するという方式でコードを生成する。避難用のメモリは事前にどのくらい必要となるかを計算しておく。コード生成器の具体的な説明は次の最適化のところで説明する。

5. 必須課題を越えて行った部分に関する詳細．どうように拡張したか，どのように作ったかな

ど.
a. 最適化
目標として、メモリへのロード・ストアの回数を最小化するという局所的な最適化であった。それを実現する方法は構文木のすべてのノードに、そのノードがルート（根）となるサブツリー（部分木）に対応する計算に必要となるレジスタの数を保存するのである。以下に必要なレジスタの数を求めるアルゴリズムを示す。

アルゴリズム
* ノードnが葉である場合
　　右の葉であれば、n ← 0
　　左の葉であれば、n ← 1
* ノードnが一つの子供n1をもつ場合
　　n ← n1
* ノードnが二つの子供n1, n2をもつ場合
　　もしn1 = n2であれば、n ← n1 + 1
　　そうでなければ、n ← max(n1, n2)
例として、図2にアルゴリズムの実行結果を示す。

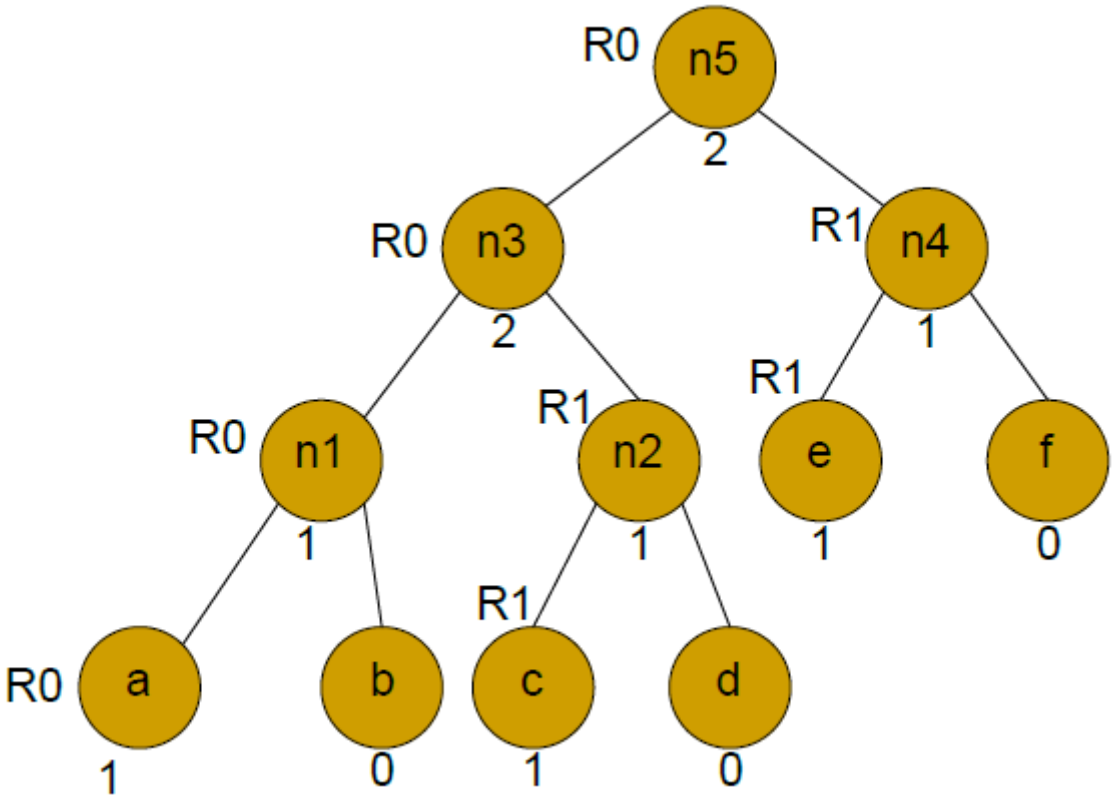

図2 レジスタの数の最小化

コード生成を行う前に三つのレジスタのスタックregと避難用のメモリのスタックmemを用意する。計算結果は常にスタックの先頭に保存し、必要な場合はスタックの先頭とその一つ前の要素を入れ替える。以下にコード生成の手順を示す。

コード生成の手順
構文木をルートから辿り始め、現在のノードnに注目し、いくつかの場合を分ける。
* ノードnが左の葉の場合
　　load n top_stack(reg)
* ノードnが二つの子供n1, n2をもつ場合
(1) n2 = 0の場合
　　cogen(n1)
　　op n2 top_stack(reg)
(2) n2 > n1 && n1 < 3の場合
　　cogen(n2)
　　r = pop_stack(reg)
　　cogen(n1)
　　op r top_stack(reg)

```
        push_stack(reg, r)
        swap_stack(reg)
(3) n1 >= n2 && n2 < 3の場合
        cogen(n1)
        r = pop_stack(reg)
        cogen(n2)
        op top_stack(reg) r
        push_stack(reg, r)
(4) n1 >= 3 && n2 >= 3の場合
        cogen(n2)
        t = pop_stack(mem)
        load top_stack(reg) t
        cogen(n1)
        op t top_stack(reg)
        push_stack(mem, t)
```

このように構文木を辿りながらコードを生成していく。

また、if文とwhile文の条件式Eの計算において、工夫した点について述べる。Eは多くの場合では以下のいずれかの形を取ることがわかる。

(1) E = const(定数)の場合、その定数を見て実行するかどうか決める。レジスタなどをまったく使用しない。

(2) E = x (+|-|==|!=|>|<|) yの場合、一つの比較命令で実行できる。但し、x + yの場合だけyを反転する必要がある。

(3) E = x * yの場合、x, yをそれぞれゼロと比較してジャンプする。

(4) E = x, !xの場合、同様にxをゼロと比較するだけである。

以上で、最適化についてまとめた。

b. for文
C0文法のstatementにfor-statementを
for'('expr;expr;expr')''{' stmt '}'
と定義し、追加した。
コード生成はwhile文を参考にした。

c. コメントアウト
tokenizer.cで/のあとに/が来た場合は¥nまで読み飛ばす。/の後に*が来たときは次に*/の連続が来るまで読み飛ばすようにした。

d. 型検査
まず、syntree_infoの中にint type;を定義し、enum{ TYPE_INT, TYPE_INT_P, TYPE_VOID, }type;の値を格納するようにした。
parser部分にて、intの次の字句が*だった場合は宣言された変数、関数の返り値がポインタであると定義した。
parse時に変数宣言部分で各変数の型を記憶していき、定数はINT_VOIDとして型を記憶していく。

mk_expr時に、変数や式の型を引数に取り、左辺の型を引き継いでいく。また、左辺と右辺の方がTYPE_INTとTYPE_INT_Pで競合すれば、警告を出すようにした。

e. ポインタ
型検査の返り値よりポインタを特定し、syntreeにポインタの式ノードなどを追加しました。また、cogen_exprにもポインタの部分を追加しましたけど、Tanくんのコードに合わせないので失敗しました。

6. チームでの役割分担
tokenizer,parserまでは各自作成後、一人のものを提出。
cogenについては基本部分はTanくんが作成。他二人は追加機能を担当。
発表用スライド、最終レポートは全員で作成。

7.課題に対する感想など
構造体を実用的に使ったプログラミングを久しぶりにしたのでとても勉強になりました。実験グ

ループのメンバーが優秀だったので教わりながらのコーディングとなりましたが、なかなかボリュームがあり、これほど長いコードを書くのは初めてでした。コンパイラの動作原理も知ることができたのでとてもよかったです。

コンパイラの実験は予想以上楽しかったです。この 12 日間は充実していました。また、時間があれば、コンパイラをさらに高速化するために様々なアルゴリズムを実装したいと思っています。C 言語でコンパイラのプログラムを書くと、基本的データ構造（リスト、スタックなど）をいちいち作らないとならないので、もし他の言語（C++など）で書くことができれば、少し楽になるではないかと思いました。